1.跳出循环

JavaScript 参考 - JavaScript | MDN (mozilla.org)

JavaScript 闭包-JavaScript中文网-JavaScript教程资源分享门户 (javascriptcn.com)

```
 //计算1-50之间，除去能被5整除的所有整数的和
var sum=0;
   for(var i = 0; i < 50; i++){
     if(i % 5 === 0) {
          console.log(i);
          continue; //结束当前循环，继续下一次循环
     }
     sum +=i;
   }
   console.log(sum);
```

2.跳出循环

```
//计算1-50之间的整数的和，如果遇到能被10整除的数就不在往下进行
var sum = 0;
for (var i = 0; i < 50; i++){
    sum += i;
    if(i % 10 === 0){
        console.log(i);
        break;  //结束循环体
    }
}
    console.log(sum);
```

break和continue：

1. break

- break关键字可以用来退出switch或循环语句 **跳出整个循环**
- break关键字，会立即终止离他最近的那个循环语句

2. continue

- continue关键字立即跳出本次循环，继续下一次循环（本次循环体中 continue 之后的代码就会少执行一次）
- continue也是默认只会对离他最近的循环循环起作用

3.跳出循环 -- indexof应用

```
indexof获取元素5的索引
var arr = [1, 2, 5, 6, 90, 46];
Array.prototype.indexof = function(item,ind){
    var ind = -1;
    for (var i = 0; i<this.length; i++){
        if (this[i] === item){
            ind = i;
            break;
        }
    }
    return ind;
}
    console.log(arr.indexof(5));
```

4 while循环

```
while(条件表达式){
  //循环体代码
}
```

执行思路：

- 先执行条件表达式，如果结果为 true，则执行循环体代码；如果为 false，则退出循环，执行后面代码；
- 执行循环体代码；
- 循环体代码执行完毕后，程序会继续判断执行条件表达式，如条件仍为true，则会继续执行循环体，直到循环条件为 false 时，整个循环过程才会结束

注意：

- 使用 while 循环时一定要注意，它必须要有退出条件，否则会出现死循环
- while 循环和 for 循环的不同之处在于 while 循环可以做较为复杂的条件判断

```
//for循环练习
//计算100以内偶数的和，除22,44,66,88之外
var sum = 0;
var outArray = [22,44,66,88];
for (var i = 0; i <= 100; i += 2) {
```

```
    //不要在for循环里面声明
    console.log(i);
    //出去 [22,44,66,88]
    if (outArray.indexOf(i) >= 0) continue;
    //求和
    sum += i;
    }
    console.log(sum);
```

```
//while循环练习
//1-100的整数和为
var i = 1;
var sum = 0;
   while (i <= 100) {
        sum += i;
        i++;
        }
   console.log('1-100的整数和为' + sum);
```

```

var list = [{ a: 1, e:30 }, { b: 20, d:50 }, { a: 2, b: 3, e:25 }];
        //原始数据1处理后的数据为:{a:3，b:5}
var obj = {};
   for (var ind = 0; ind < list.length; ind++) {
     console.log(list[ind]);//依次打印list里面的数组元素
        for(var k in list[ind]){
            console.log(k, list[ind][k]); //[]解析变量k，k表示属性，list[ind][k]表示属性值
            //如果属性已经存在，就累加
            if(obj[k]){
                obj[k] += list[ind][k];
            }else{
             //如果该属性不存在，那么就进行初始化
                obj[k] = list[ind][k];
              }
         }
    }
     console.log(obj);
```

```
        console.log(obj);
```

```
//while和indexof计算o出现的次数
var str = 'asfsdfoadssafdsoaoodfosdfo';
var count = 0, start = 0;
    while(str.indexOf('o', start) > -1){
         count++;
         start = str.indexOf('o', start) + 1;
   }
     console.log(count); //6
```

5 do while循环

```
do {
  //循环体代码---->条件表达式为true的时候重复执行循环代码
}while(条件表达式);
```

执行思路：

1. 先执行一次循环体代码
2. 再执行条件表达式，如果结果为true，则继续执行循环体代码，如果为false，则退出循环，继续执行后面的代码
3. 先执行再判断循环体，**所以do while循环语句至少会执行一次循环体代码**

函数

使用函数声明来创建一个函数：

```
function 函数名([形参1, 形参2...形参N]) {
    语句...
}
// 调用函数
函数名();
```

```
/**
1、使用function声明函数；
```

```
2、dateFormat是函数名；
3、()里面的date是形参；
4、{}是函数体；
*/
function dateFormat(date){
    //转换成事件对象是为了更方便操作
     var dateObj = new Date(date);
         return `${dateObj.getFullYear()}-${(dateObj.getMonth() + 1)
         .toString().padStart(2, '0')}-${dateObj.getDate().toString()
         .padStart(2, '0')} ${dateObj.getHours().toString()
         .padStart(2, '0')}:${dateObj.getMinutes().toString().padStart(2, '0')}
         :${dateObj.getSeconds().toString().padStart(2, '0')}`;
         //toString()把数字转换成字符串，在用padStart()填充当前字符串
         console.log(500);
    }
    console.log(dateFormat(new Date()));
    console.log(dateFormat('Thu Aug 05 2021 15:20:30 GMT+0800'));
```

使用函数表达式来创建一个函数:

```
var 函数名 = function([形参1, 形参2...形参N]) {
     语句...
  };
 // 调用函数
  函数名();
```

自运行函数:

JS 自执行函数原理及用法

主要介绍了JS 自执行函数原理及技巧,文中通过示例代码介绍的非常详细，对大家的学习或者工作具有一定的参考学习价值,需要的朋友可以参考下

js自执行函数，听到这个名字，首先会联想到函数。接下来，我来定义一个函数：

```
function aaa(a,b){
  return sum = a + b
}
```

定义了一个名为aaa的函数，在里面可以计算两个数的和。如果想执行它，就必须得调用它，并且还得给它传参：

```
var aa = aaa(1,2)
```

这样就实现了一个函数的定义与调用，通过console.log我们可以看到sum实现了两个数的相加。

自执行函数是什么？自执行函数就是当它被定义出来，就会自动执行的函数。不需要调用，传参也很方便。就上面的函数，用自执行函数定义就是这样：

```
(function aaa(a,b){
  return sum = a + b
;})(1,2)
```

通过控制台可以发现sum实现了两个数的相加。

自执行函数有三种写法：

1.(function (“ 参数 ”) { " 函数方法 " ; }) (“ 给参数传的值 ”)

2.(function (" 参数 ") { “ 函数方法 ” ; } (" 给参数传的值 "))

3.! function (" 参数 ") { " 函数方法 " ; } (" 给参数传的值 ")

第三种!可以换作其他运算符或者void。

自执行函数是很自私的，它的内部可以访问全局变量。但是除了自执行函数自身内部，是无法访问它的。例：

```
function aaa(a1,b1){
  return sum1 = a1 + b1
},
(function bbb(a2,b2){
  return sum2 = a2 + b2
;}(),
console.log(aaa)
console.log(bbb)
```

这是一个函数与一个自执行函数，输出这两个函数会发现：函数aaa被全部打印出来，而bbb则报错。自执行函数相当于一个瓶口朝下的杯子，当定义它的时候，它会倾斜，把杯口露出来，吸收外面的新鲜空气；当它执行完毕，杯口不再外露，紧闭起来，与外界再无关联。

---

arguments的使用：

当我们不确定有多少个参数传递的时候，可以用 arguments 来获取。在 JavaScript 中，arguments 实际上它是当前函数的一个内置对象。所有函数都内置了一个 arguments 对象，arguments 对象中存储了传递的所有实参。

```
1、arguments是接收过来的所有的实参；
2、arguments是一个类数组，可以进行遍历。类数组具有以下特点
    ①：具有 length 属性
    ②：按索引方式储存数据
    ③：不具有数组的 push ， pop 等方法
3、使用Array.from把argument转换成真正的数组；
```

```
4、就可以使用reduce：

第1次执行：  p就是初始化的0，v就是第1个元素
第2次执行：  p就是函数上一次调用时候的返回值，v就是第2个元素

 var sum = function () {
     return Array.from(arguments).reduce(function (p, v) {
         return p * v;
     }, 1);
  }
     console.log(sum(10, 20, 30, 40));

 //利用函数求任意个数的最大值
 function maxValue() {
     var max = arguments[0];
     for (var i = 0; i < arguments.length; i++) {
         if (max < arguments[i]) {
             max = arguments[i];
         }
     }
     return max;
}
console.log(maxValue(2, 4, 5, 9)); // 9
console.log(maxValue(12, 4, 9)); // 12
```

匿名函数：

```
/**函数后面跟上小括号表示函数的执行*/
/**形成一个封闭的空间*/
(function (a) {
     var a = 200;
     console.log(a);
     function add() {
    }
})(100);
```

this指向的情况，取决于函数调用的方式有哪些：

- 通过函数名()直接调用的：this指向window

```
function myobj(){
    var myname = "ABC";
    console.log(this);
    console.log(this.myname);
}
myobj(); //等价于 window.myobj();
//函数执行中this指向的并不是函数本身（函数也是对象），而是调用它的对象，浏览器
//的全局变量是window，所以这里的this指向window，当然window.mynmme为undifined
```

- 通过 对象.函数名() 调用的：this指向这个对象；

```
var myname = "abc";
function myobj(){
    var myname = "ABC";
    console.log(this);
    console.log(this.myname);
}
var a = {
    fun:myobj, //将函数myobj 赋值给a.fun
    myname:"我是a的name"
};
myobj();
a.fun();
//直接调用myobj();实际上是window.myobj(),所以调用myobj的是全局window，
//所以这里的this指向window， 而将myobj赋值给a对象的fun属性后，调用a.fun(),
//就是a来调用myobj（），所以this指向a
```

```
//对象.对象.函数名调用
var myname = "lee";
function myobj(){
    var myname = "LEE";
    console.log(this);//
    console.log(this.myname);//
}
var a = {
    myname:"我是a的name",
    fun:myobj,
```

```
    b:{
        myname:"我是b的name",
        fun:myobj
    }
};
myobj();//Window lee
a.fun();//{myname: '我是a的name', b: {…}, fun: ƒ}   我是a的name
a.b.fun();//{myname: '我是b的name', fun: ƒ}  我是b的name
```

- 函数作为数组的一个元素，通过数组下标调用的：this指向这个数组

```
var myname = "我是window的name";
 function myobj(){
     var myname = "ABC";
     console.log(this);
     console.log(this.myname);
     console.log(this[0].myname);//打印数组第0个元素myname属性
 }
 var myarr = [{myname:"我是myarr[0]的name"},myobj];//新建以数组对象
 myarr.myname = "我是myarr的name"; //为数组对象添加myname属性
 myarr[1]();//数组下标方式调用函数
```

- 函数作为window内置函数的回调函数调用时：this指向window.setTimeout(func,XXms);setInterval(func,XXms)等

```
var myname = "222";
function myobj(){
     var myname = "111";
     console.log(this);
     console.log(this.myname);
 }
  setTimeout(() => {
   myobj();
 }, 1000);
 //222
```

- 函数作为构造函数，用new关键字调用时：this指向的是new出的新对象

```
var myname = "333";
```

```
function Myobj(){  //构造函数通常首字母大写
    this.myname = "444";
    console.log(this);
    console.log(this.myname);
}
var mobj = new Myobj();
//444
```

通过函数指定，用apply()、call()、bind() 方法指定this

call()和apply():

这两个方法都是函数对象的方法，需要通过函数对象来调用

当对函数调用call()和apply()都会调用函数执行

```
var obj1 = {
    name: "obj1"
};
var obj2 = {
    name:"obj2"
}
function fun(){
    console.log(this.name);
}
fun.call(obj1); // obj1
fun.call(obj2); // obj2
```

调用call()和apply()可以将一个对象指定为第一个参数，此时这个对象将会成为函数执行时的this

- call() 方法使用一个指定的 this 值和单独给出的一个或多个参数来调用一个函数；当第一个参数为null、undefined的时候，默认指向window。
- apply() 方法调用一个具有给定this值的函数，以及以一个数组（或类数组对象）的形式提供的参数；当第一个参数为null、undefined的时候，默认指向window

区别：call() 方法接受的是一个参数列表，而 apply() 方法接受的是一个包含多个参数的数组

bind()：

方法创建一个新的函数，在 bind() 被调用时，这个新函数的 this 被指定为 bind() 的第一个参数，而其余参数将作为新函数的参数，供调用时使用

```
var obj = {
    a: 100,
```

```
    fn: function (m, n) {
    console.log(this.a + m + n);
    }
};

//方法劫持
var obj1 = {
    a: 20
   };

//call():
/**
call可以改变this的指向
this指向call的第一个参数obj1
会立即执行
参数时列表形式
不传参数，或者第一个参数是null或undefined，this都指向window
*/
   obj.fn.call(obj1, 30, 40); //90
   obj.fn.call(null, 30, 40); //NaN

//apply():
/**
apply可以改变this的指向
this指向apply的第一个参数obj1
会立即执行
参数时数组形式
*/
   obj.fn.apply(obj1, [40, 50]); //110


//bind() 方法不会立即执行，而是返回一个改变了上下文 this 后的函数。
/**bind主要用于改变this的指向
this指向bind的第一个参数obj1
不会立即执行，需添加()来调用才会执行
参数时列表形式
*/
    console.log(obj.fn.bind(obj1, 60, 70)); //没有调用
    console.log(obj.fn.bind(obj1, 60, 70)()); //150 undefined

```

```
var obj2 = obj.fn.bind(obj1, 10, 10)();
    console.log(obj2);
```

- 参数的使用

```
function fn(a, b, c) {
    console.log(a, b, c);
}
var fn1 = fn.bind(null, 'yes');

fn('A', 'B', 'C');   // A B C
fn1('A', 'B', 'C');    // yes A B
fn1('B', 'C');       // yes B C
fn.call(null, 'yes');    // yes undefined undefined
fn.apply(null, [1, 2, 3]);  // 1 2 3
fn.apply(null, [1, 2, 3, 4, 5]); // 1 2 3
```

call 是把第二个及以后的参数作为 fn ()的实参传进去；而 fn1 ()的实参实则是在 bind 中参数的基础上再往后排；apply()把[1, 2, 3]作为fn ()的实参传进去，多余参数不会传进去

用bind方法实现函数柯里化:

在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数 (最初函数的第一个参数) 的函数，并且返回接受余下的参数且返回结果的新函数的技术。

```
var add = function(x) {
    return function(y) {
        return x + y;
   };
};

var add1 = add(1);
var add2 = add(10);
add1(2);// 3
add2(2);// 12
```

在低版本浏览器没有 bind 方法:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function () {
        var self = this,                        // 保存原函数
```

```
            context = [].shift.call(arguments), // 保存需要绑定的this上下文
            items = [].slice.call(arguments);    // 剩余的参数转为数组
        return function () {                     // 返回一个新函数
            self.apply(context, [].concat.call(items, [].slice.call(arguments)));
        }
    }
}
```

call、apply、bind方法应用:

- 求数组中的最大和最小值

```
var arr = [1, 2, 3, 89, 46]
var arr1 = Math.max.call(arr, ...arr) //...表示扩展运算符
    console.log(arr1); //89
var max = Math.max.apply(null, arr)
    console.log(max); //89
var min = Math.min.apply(null, arr) //1
    console.log(min); //1
```

- 将类数组转化为数组

```
var arr = Array.prototype.slice.call(Arr);
```

- 数组追加

```
var arr1 = [1,2,3];
var arr2 = [4,5,6];
var arr3 = [].push.apply(arr1, arr2);
    console.log(arr3); //6
    console.log(arr1); //[1, 2, 3, 4, 5, 6]
    console.log(arr2); // [4,5,6]
```

- 判断变量类型

```
function isArray(obj){
    return Object.prototype.toString.call(obj) == '[object Array]';
}
    console.log(isArray([]));// true
    console.log(isArray('yes'));// false
```

- **利用call和apply做继承（方法劫持）**

```
function Person(name,age){
    // 这里的this都指向实例
    this.name = name
    this.age = age
    this.say = function(){
        console.log(this.age)
    }
}
function Female(){
    Person.apply(this,arguments)//将父元素所有方法在这里执行一遍就继承了
}
var foo = new Female('abc',2)
    console.log(foo); //Female {name: 'abc', age: 2, say: f}
```

- 箭头函数中的this问题，指向为定义时的this，而不是执行时的this